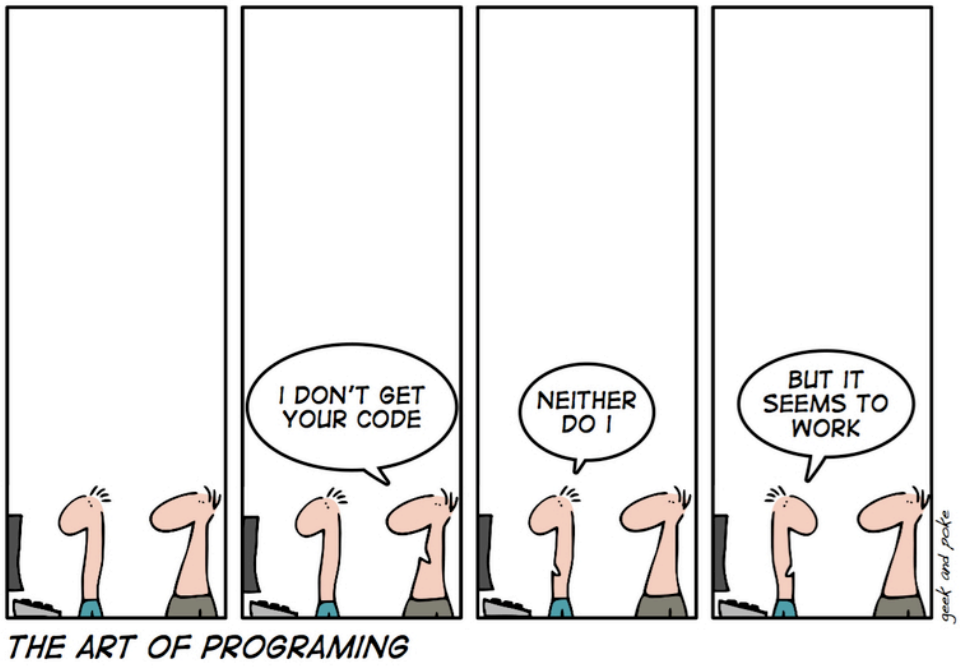

geekandpoke.typepad.com

It's funny because it's true.

Writing clear code

Karl Broman

Biostatistics & Medical Informatics, UW–Madison

kbroman.org
github.com/kbroman
@kwbroman

Clear code is more likely to be correct.

Clear code is easier to use.

Clear code is easier to maintain.

Clear code is easier to extend.

Basic principles

- Code that works
  - No bugs; efficiency is secondary (or tertiary)
- Readable
  - Fixable; extendible
- Reusable
  - Modular; reasonably general
- Reproducible
  - Re-runnable
- Think before you code
  - More thought $\Longrightarrow$ fewer bugs/re-writes
- Learn from others' code
  - R itself; key R packages

Our first goal should be to get the right answer.

But we also want to write code that we (and others) will be able to use again. Efficiency is way down on the list of needs.

Write clearly, and re-write for greater clarity.

Break things down into small, re-usable functions, written in a general way, but not too general. You want to actually solve a particular problem.

Step 1: stop and think.

Step 2: write re-usable functions rather than a script.

By reproducible/re-runnable, I mean: don't have a bunch of code from which you copy-and-paste in a special way.

Write programs for people, not computers

Wilson et al. (2014) PLoS Biol 12:e1001745

The computer doesn't care about comments, or style, or the naming of things. But users and collaborators (and yourself six months from now) will.

Break code into small functions

```
get_grid_index <-
function(vec, step)
{
  grid <- seq(min(vec), max(vec), by=step)
  index <- match(grid, vec)

  if(any(is.na(index)))
    index <- sapply(grid, function(a,b) {
      d <- abs(a-b)
      wh <- which(d==min(d))
      if(length(wh)>1) wh <- sample(wh, 1)
      wh
      }, vec)

  index
}
```


This is a bit of code that was originally part of a larger function.

Separated as a function, the code is more likely to be reused. And the smaller and more focused the function, the more likely you'll use it again.

Avoid having scripts that are long streams of code. Write functions. Give each function a single, focused task. Break long functions into pieces.

Comment on the input and output; ideally, these are obvious from the names.

What might be improved about this function?

Break code into small functions

```
sampleone <-
function(vec)
  ifelse(length(vec)==1, vec, sample(vec, 1))

get_grid_index <-
function(vec, step)
{
  grid <- seq(min(vec), max(vec), by=step)
  index <- match(grid, vec)

  if(any(is.na(index)))
    index <- sapply(grid, function(a,b) {
        d <- abs(a-b)
        sampleone(which(d == min(d)))
      }, vec)

  index
}
```


Ideally, a function does only one thing.

Pulling out the `sampleone` code as a separate function makes the body of `get_grid_index` a bit simpler.

Is the `ifelse` line clear? Would it be better to just write that function with `if` and `else`?

Could `get_grid_index` be improved further?

Clarity over efficiency

```
sampleone <-
function(vec)
  ifelse(length(vec)==1, vec, sample(vec, 1))

get_grid_index <-
function(vec, step)
{
  grid <- seq(min(vec), max(vec), by=step)
  index <- match(grid, vec)

  if(any(is.na(index))) {
    for(i in seq(along=grid)) {
      d <- abs(grid[i] - vec)
      index[i] <- sampleone(which(d==min(d)))
    }
  }

  index
}
```


The use of `sapply` is a bit confusing and hard to follow. I think it's a bit more clear to use a `for` loop.

Whether `apply`-type functions are more readable or less readable depends on the complexity of the function that is applied but also on the experience of the reader.

One last change

```
sampleone <-
function(vec)
  ifelse(length(vec)==1, vec, sample(vec, 1))

get_grid_index <-
function(vec, step)
{
  grid <- seq(min(vec), max(vec), by=step)
  index <- match(grid, vec)

  missing <- is.na(index)

  if(any(missing)) {
    for(i in which(missing)) {
      d <- abs(grid[i] - vec)
      index[i] <- sampleone(which(d==min(d)))
    }
  }

  index
}
```


One last change: The loop really only needs to be over the parts of `index` that are missing.

Of course, I should also add a comment for each function, to explain the input and output.

Another example

```
# rmvn: simulate from multivariate normal distribution
rmvn <-
function(n, mu=0, V=diag(rep(1, length(mu))))
{
  p <- length(mu)

  if(any(dim(V) != p))
    stop("Dimension problem!")

  D <- chol(V)

  matrix(rnorm(n*p),ncol=p) %*% D + rep(mu,each=n)
}
```


I'm often wanting to simulate from a multivariate normal distribution. It's not hard, but you have to remember: do you do `Z %*% D` or `D %*% Z`? And in any case, it's a lot easier to just write `rmvn(n, mu, V)`.

Note the use of default values.

Further examples

```
# colors from blue to red
revrainbow <-
function(n=256, ...)
  rev(rainbow(start=0, end=2/3, n=n, ...))



# move values above/below quantiles to those quantiles
winsorize <-
function(vec, q=0.006)
{
  lohi <- quantile(vec, c(q, 1-q), na.rm=TRUE)
  if(diff(lohi) < 0)
    lohi <- rev(lohi)

  vec[!is.na(vec) & vec < lohi[1]] <- lohi[1]
  vec[!is.na(vec) & vec > lohi[2]] <- lohi[2]

  vec
}
```


If there's a bit of code that you write a lot, turn it into a function.

I can't tell you how many times I typed `rev(rainbow(start=0, end=2/3, n=256))` before I thought to make it a function. (And I can't tell you how many times I've typed it since, having forgotten that I wrote the function.)

(And really, I should forget about `revrainbow`; such rainbow color scales are generally considered a bad idea.)

I learned about Winsorization relatively recently; it's a really useful technique to deal with possible outliers. It's not hard. But how much easier to just write `winsorize()`!

Writing functions

- Break large tasks into small units.
  - Make each discrete unit a function.
- If you write the same code more than once, make it a function.
- If a line/block of code is complicated, make it a function.

Modularity is really important for readability.

No long streams of code; rather, a short set of function calls.

Don’t repeat yourself (or others)

- Avoid having repeated blocks of code.
- Create functions, and call those functions repeatedly.
- This is easier to maintain.
  - If something needs to be fixed/revised, you just have to do it the one time.
- Look at others’ libraries/packages.
  - Don’t write what others have already written (especially if they’ve done it better than you would have).

Don’t repeat yourself (DRY) is one of the more important concepts for programmers.

I’m particularly bad at making use of others’ code.

Don't make things too specific

- Write code that is a bit more general than your specific data
  - Don't assume particular data dimensions.
  - Don't forget about the possibility of missing values (even if your data doesn't have any).
  - Aim for re-use.
- Use function arguments
  - Don't assume particular data file names
  - Don't hard-code tuning parameters
  - R scripts can take command-line arguments:

```
Rscript myscript.R input_file
output_file
args <- commandArgs(TRUE)
```


Don't make things too general, either. You want to actually solve your particular problem.

You don't want to have to edit the code for different data.

We will write scripts. But make them a bit more general by allowing command-line arguments to specify input and output file names. You can always include defaults, for your particular files. Even better is to have these specified in your `Makefile`.

No global variables, ever!

- Don't refer directly to objects in your workspace.
- If a function needs something, pass it as an argument.
- (But what about really big data sets?)

Global variables make code unreadable.

Global variables make code difficult to reuse.

It turns out that, when you pass data into a function, R doesn't actually make a copy. If your function makes any change to the data object, then it will make a copy. But you shouldn't be afraid to pass really big data sets into and between functions.

No magic numbers

- Name numbers and use the names

```
max_iter <- 1000
tol_convergence <- 0.0001
```

- Even better: include them as function arguments

You'll later ask yourself, "Why `12`?"

When you include such things as function arguments, include reasonable default values.

Indent!

```
# move values above/below quantiles to those quantiles
winsorize <-
function(vec, q=0.006)
{
lohi <- quantile(vec, c(q, 1-q), na.rm=TRUE)
if(diff(lohi) < 0)
lohi <- rev(lohi)
vec[!is.na(vec) & vec < lohi[1]] <- lohi[1]
vec[!is.na(vec) & vec > lohi[2]] <- lohi[2]
vec
}
```


I taught part of a course on statistical programming at Johns Hopkins. It included a lecture like this one. Still, many students submitted their programming assignment flush left.

Most people recommend indenting by 4 spaces. Still, I prefer 2.
(4 spaces are easier to track.)

Don't use tabs, as they get messed up when moving between computers.

Use white space

```
# move values above/below quantiles to those quantiles
winsorize<-function(vec,q=0.006)
{lohi<-quantile(vec,c(q,1-q),na.rm=TRUE)
if(diff(lohi)<0)lohi<-rev(lohi)
vec[!is.na(vec)&vec<lohi[1]]<-lohi[1]
vec[!is.na(vec)&vec>lohi[2]]<-lohi[2]
vec}
```


The computer doesn't care about white space, but people do.

Don’t let lines get too long

```
get_grid_index <-
function(vec, step)
{
  grid <- seq(min(vec), max(vec), by=step)
  index <- match(grid, vec)

  if(any(is.na(index)))
    index <- sapply(grid, function(a,b) { d <- abs(a-b); sampleone(whi

  index
}
```


$<$ 72 characters per line

Use parentheses to avoid ambiguity

```
if( (ndraws1==1) && (ndraws2>1) ) {

 ...

}

leftval <- which( (map - start) <=0 )
```


Even if they aren't necessary, they can be helpful to you (or others) later.

Names: meaningful

- Make names descriptive but concise
- Avoid `tmp1`, `tmp2`, ...
- Only use `i`, `j`, `x`, `y` in the simplest situations
- If a function is named `fv`, what might it do?
- If an object is called `nms`, what could it be?
- Functions as verbs; objects as nouns

I'm terrible at coming up with good names. But it's really important.

Descriptive names make code self-documenting.

If functions are named by what they do, you may not need to explain further.

If the object names are meaningful, the code will be readable as it stands. If the objects are named `g`, `g2`, and `gg`, it'll need a lot of comments.

I admit, I'm a terrible offender in this regard, especially when I'm coding in a hurry. Do as I say, not as I do.

Names: consistent

- `markers` vs `mnames`
- `camelCase` vs. `pothole_case`
- `nind` vs `n.var`
- If a function/object has one of these, there shouldn't be a function/object with the other.

Consistency makes the names easier to remember.

If the names are easier to remember, you're less likely to use the wrong name and so introduce a bug.

And, of course, other users will appreciate the consistency.

Names: avoid confusion

- Don't use both `total` and `totals`
- Don't use both `n.cluster` and `n.clusters`
- Don't use both `result` and `results`
- Don't use both `Mat` and `mat`
- Don't use both `g` and `gg`

Don't use similar but different names, in the same function or across functions.

It's confusing, and it's prone to bugs from mistyping.

Comments

- Comment the tricky bits and the major sections
- Don't belabor the obvious
- Don't comment bad code; rewrite it
- Document the input/output and purpose, not the mechanics
- Don't contradict the code
  - this happens if you revise the code but don't revise the related comments
- Comment code as you are writing it (or before)
- Plan to spend 1/4 of your time commenting

Some things are tricky and deserve explanation, so that when you come back to it, you have a guide.

But in many cases, code can be rewritten to be clear and self-documenting.

Commenting takes time. And no one ever goes back and adds comments later. It's another one of those "invest for the future" type of things.

Error/warning messages

- Explain what's wrong (and where)
  - `error("nrow(X) != nrow(Y)")`
- Suggest corrective action
  - `"You need to first run calc.genoprob()."`
- Give details
  - `"nrow(X) (", nrX, ") != nrow(Y) (", nrY, ")"`
- Don't give error/warning messages that users won't understand.
  - `X'X is singular.`
- Don't let users do something stupid without warning
- Include error checking even in personal code.

Meaningful error messages are hard.

Writing them to give detailed information takes time.

And of course you wouldn't be making these errors, right?

But remember that you, six months from now, will be like a whole new person.

Check data integrity

- Check that the input is as expected, or give warnings/errors.
- Write these in the first pass (though they're dull).
  - You may not remember your assumptions later
- These are useful for documenting the assumptions.

You can't include enough such checks.

Hadley Wickham's `assertthat` package is great for this. We'll talk about it in a couple of weeks.

Program organization

- Break code into separate files (say 300 lines?)
- Each file includes related functions
- Files should be named meaningfully
- Include a brief comment at the top.

My R/qtl package has some really long files, like `util.R`, which seemed like a good idea at the time. I have to use a lot of `grep` to find things. And there are plenty of functions in there that I have totally forgotten about.

I used to make really long boiler-plate comments at the top of each file. That means that you always have to page down to get to the code. Now I try to write really minimal comments.

Create an R package!

- Make a personal package with bits of your own code
- Mine is R/broman, github.com/kbroman/broman

```
# qqline corresponding to qqplot
qqline2 <- function(x, y, probs = c(0.25, 0.75), qtype = 7, ...)
{
  stopifnot(length(probs) == 2)
  x <- quantile(x, probs, names=FALSE, type=qtype, na.rm = TRUE)
  y <- quantile(y, probs, names=FALSE, type=qtype, na.rm = TRUE)
  slope <- diff(y)/diff(x)
  int <- y[1L] - slope*x[1L]
  abline(int, slope, ...)
  invisible(c(intercept=int, slope=slope))
}
```


R packages are great for encapsulating and distributing R code with documentation.

Every R programmer should have a personal package containing the various functions that they use day-to-day.

We'll talk about writing R packages next week. It's really rather easy.

I give an example here of one function in my R/broman package. Some of the earlier examples are in this package.

Complex data objects

- Keep disparate data together in a more complex structure.
  - lists in R
  - I also like to hide things in object attributes
- It's easier to pass such objects between functions
- Consider object-oriented programming

Complex data objects allow you to think about complex data as a single thing. And they make it easier to pass the data into a function.

R has two systems for object-oriented programming: S3 and S4. They can make your code easier to use and understand. But they can also make things more opaque and harder to understand.

Avoiding bugs

- Learn to type well.
- Think before you type.
- Consider commenting before coding.
- Code defensively
  - Handle cases that ”can’t happen”
- Code simply and clearly
- Use modularity to advantage
- Think through all special cases
- Don’t be in too much of a hurry

Writing clear code will make it easier to find bugs, but we also want to avoid introducing bugs in the first place. These are my strategies; They are useful when applied.

Basic principles

- Code that works
  - No bugs; efficiency is secondary (or tertiary)
- Readable
  - Fixable; extendible
- Reusable
  - Modular; reasonably general
- Reproducible
  - Re-runnable
- Think before you code
  - More thought $\Longrightarrow$ fewer bugs/re-writes
- Learn from others' code
  - R itself; key R packages

It seems useful to repeat these basic principles.

Summary

- Get the correct answers.
- Find a clear style and stick to it.
- Plan for the future.
- Be organized.
- Don't be too hurried.
- Learn from others.

Our primary goal, as scientific programmers, is to get the right answer.

We all write differently. That's okay. But be consistent. If every function is named in a different way, it will be that much harder to remember.

Plan for the future: write code that is readable and easy to maintain. And write things in a modular and reasonably general way, so that you (or others) will be more likely to be able to re-use your work.

Be organized. Well-organized software is easier to understand and maintain.

It's when I'm hurried that I write crap code. It means more work later.

There are a lot of great programmers out there; read their code and learn from it.